FALTAN LAS OPINIONES

Tarea 2

Grupo EC1-5

Algoritmo básico con estrategias no informadas

* Enlace a GitHub: <https://github.com/Agustin6199/EC1-5>
* Componentes del grupo:
  + José Otero Aguado
  + Agustín Mora Acosta
  + Germán Rolando León
* Explicación de cada clase:

Es necesario saber qué estructura tiene el programa para saber cómo funciona a la hora de resolver el programa por lo que, a continuación, explicamos las clases y qué contienen cada una de ellas.

* **Clase Búsqueda**

Las primeras líneas de código de esta clase son las llamadas import necesarias para poder usar las excepciones predefinidas en la API de Java y las estructuras para interactuar con las estructuras de datos como pueden ser los Arraylist o pilas.

Después de la definición del modificador de acceso de la clase junto al nombre, se crea una clase de tipo *enum* en la que se definen los tipos de búsqueda que va a recoger nuestro algoritmo.

Esta clase contiene nuestro método ***main*** para poder ejecutar el programa y resolver el problema dado. Dentro de este método *main* lo primero que nos encontramos es la creación de un objeto problema que se va a encargar de, una vez leído el *json*, recoger el estado inicial del problema que, a su vez recogerá el cubo a resolver. Después de leer el *json* y crear el problema se hace una primera lectura del tiempo actual para, seguidamente de ejecutar el método que contiene el algoritmo de búsqueda volver a medir el tiempo y así saber cuánto ha tardado. Como se ha mencionado antes se hace una llamada al método ***Busqueda\_Acotada*** que devolverá un valor booleano mostrando true si ha obtenido una solución y false si no obtuvo ninguna.

Centrándonos en el método de ***Busqueda\_Acotada*** devuelve un tipo boolean representando si ha encontrado solución o no y recibe por parámetros un objeto Problema que contendrá el estado inicial y en él el cubo a resolver, un objeto Estrategia que, al venir de una clase tipo *enum*, definirá el comportamiento del algoritmo y con ello el tipo de estrategia a seguir y por último, en caso de que se necesite, una profundidad máxima a la que llegar al ser una búsqueda acotada.

Las primeras líneas del método están dedicadas a la declaración de las variables que van a ser usadas, en este caso las siguientes: NodoArbol *n\_actual* se va a encargar de ir adquiriendo el valor del nodo que se esté analizando en ese momento, Frontera *frontera* tiene como objetivo guardar todos los nodos con menor f que se vaya encontrando el algoritmo a medida que vaya expandiendo los nodos, boolean sol tomando el valor inicial de false representará si el algoritmo ha hallado la solución tomando en ese caso el valor de true, HashMap que cobrará utilidad más adelante a la hora de crear la lista con los nodos del árbol y por último otro objeto NodoArbol *n\_inicial* para indicar el primer nodo que se genera a partir del que se da en el objeto problema.

En estas primeras líneas también se reinicia el ID del NodoArbol que al ser una variable estática se reiniciara para todos los objetos NodoArbol y así empezar desde cero la cuenta de los ID’s. Como primera línea antes del bucle que realizará la búsqueda se establece el nodo inicial en la frontera al ser el primer nodo que formará camino solución.

A continuación, en el mismo método, nos encontramos con un bucle while que, mientras el valor de sol sea false y mientras la frontera no esté vacía seguirá iterando. Dentro de éste lo primero que se hace es asignar a la variable n\_actual el primer nodo de la frontera, para así ir sacando uno a uno y comprobando si es solución o no. En el caso de que sea solución dejará de cumplir la condición del while por lo que se saldrá y procederá a ejecutar el siguiente if que explicaremos más adelante, en caso de que no fuese solución ejecutaría el else. Dentro de este else, hay otra estructura if que comprueba que no se haya alcanzado la máxima profundidad ya que en ese caso no tendría solución, si no se ha llegado a la máxima profundidad genera los sucesores de ese nodo (n\_actual) a través del método *Sucesores()* y pasa el resultado de éste método a otro llamado *CrearListaNodosArbol()*, debido a su complejidad vamos a pararnos a explicarlo detalladamente.

**CrearListaNodosArbol** es un método que, una vez recibido por parámetros una lista de sucesores, el nodo del cual se han desarrollado los sucesores, la profundidad máxima de esa búsqueda, la estrategia a seguir y una estructura HashMap, se encarga de generar todos los objetos NodoArbol con sus valores correspondientes de coste, profundidad, heurística, ID, f y estado podando aquellos NodoArbol que ya se hayan expandido anteriormente y tengan mayor f que ese anterior. Para lleva a cabo todas estas funciones: en el caso del coste, se suma el suyo más el que tiene el padre, en el caso de la profundidad se aumenta en uno, en la heurística se calcula mediante la entropía y ésta a su vez mediante la fórmula descrita en el campus virtual (entropía = entropía + contador[c]/(N\*N) \* math.log(contador[c]/(N\*N),6)), el ID mediante el propio constructor del NodoArbol y la f con los valores devueltos del método ***ObtainF()***. En cuanto a la poda, ésta se lleva a cabo mediante if anidados, el primero de éstos comprueba qué estrategia se está siguiendo para poder distinguir entre la profundidad y las demás ya que la f se comporta distinto, en el segundo de ellos se revisa la tabla HashMap para ver si el valor MD5 generado por ese nodo se encuentra ya en ésta, si no es así se inserta directamente ya que esto significará que no se ha generado, en cambio, si sí coincide, entra en el tercer if que se encarga de comprobar cuál de las f es mayor (en el caso de la profundidad) o menor (en el resto de estrategias) y así meterla en la HashMap o no.

Volviendo, ahora que ya sabemos el funcionamiento de CrearListaNodos, al método Busqueda\_Acotada se rellena la frontera con los nodos restantes de la poda y se haría la siguiente iteración analizando los nodos de ésta. En caso de que sí que sea solución, saldría del while al no cumplirse la condición y entraría al siguiente if. En éste se crea una pila que tiene como función albergar los objetos NodoArbol resultantes de las llamadas a getNodoPadre() de la clase NodoArbol para así encontrar el camino solución. Posteriormente se imprimen los resultados para ser acordes a las soluciones descritas y ser mejor interpretables. En caso de que no entrase al if porque después de haber recorrido todos los nodos, teniendo, por tanto, la frontera vacía, entraría en la sección else del último if e imprimiría el mensaje “No existe solución”.

* **Clase Cubo**

Las primeras líneas de código de esta clase son, como en la anterior, llamadas a import para poder usar, en este caso, las excepciones predefinidas de la API de Java y los métodos de la clase *gson* de Google para poder leer los ficheros con formato *json*.

La clase cubo comienza describiendo una serie de atributos, concretamente seis que representan cada una de las caras del cubo. Matrices de dimensión ladoxlado que guardarán los valores numéricos de cada posición de la cara.

Posteriormente se tiene una sobrecarga de constructores para poder crear el cubo a partir de un archivo *json* o directamente dado un número de lado para generar un cubo completo y resuelto.

Después de estos métodos nos encontramos con el método ***clone***en el que comienza generando un cubo resuelto de tamaño n llamado clone. El tamaño de n se halla obteniendo el tamaño de la matriz del atributo back (cara posterior del objeto cubo). Seguidamente nos encontramos con dos bucles for anidados en el que recorre todas las matrices anteriormente mencionadas asignando los valores que contienen las matrices atributo de la clase. Luego, se llama a los métodos setters para definir los valores de los atributos del objeto clone con las matrices previamente rellenadas. Para finalizar devuelve el objeto cubo *clone*.

Seguidamente nos encontramos con un bloque de código perteneciente a todos los métodos **setters** y **getters** de la clase cubo para poder asignar valores y devolver valores de los atributos de la clase, respectivamente. Un método set y otro método get por cada atributo de la clase.

Más adelante podemos ver el método ***Move***, devolviendo *void,* pero recibiendo por parámetro un *String* *movement*. En las primeras líneas del método se hacen unas comprobaciones del argumento para comprobar que es correcta la longitud y formato y posteriormente se desglosa éste en tres variables: la primera variable llamada *move\_type* es un dato de tipo String que engloba el primer carácter del argumento ya que va a indicar qué cara se va a mover, si la lateral, la cara de abajo o la posterior, la segunda variable llamada *move\_num* es el segundo carácter del argumento de tipo *int* puesto que recoge el número de cara de las n que hay por lado que se va a rotar y por último, la tercera variable llamada *upper* es de tipo *boolean* y recoge si la primera variable es mayúscula o minúscula para saber en qué sentido se debe hacer la rotación.

Dentro del método *Move* y después de recoger el valor de esas tres variables hay una estructura switch en la que según qué letra de las tres posibles (L left, D down y B back) se ha recogido en *move\_type* entra en un *case* concreto. Dentro de cada case, según si la variable booleana *upper* es true o false llamará a el método correspondiente para mover hacia un sentido u otro.

La explicación de los métodos Move con las distintas variantes según la cara son prácticamente idénticas ya que consiste en mover unas posiciones determinadas por las posiciones determinadas de otra cara, simulando el cubo en la realidad, pero en nuestro programa las posiciones son valores en una matriz. Hay dos casos concretos ya que si lo que se pide es mover cualquiera de las dos caras de los extremos lo que se va a generar es una rotación de la cara completa.

Estos dos últimos casos nos llevan al siguiente método ***Rotate\_Face***en el que básicamente lo que se implementan son dos for anidados encargados de mover las posiciones de la matriz correspondiente para dejarlas rotadas, al igual que los métodos anteriores, pero con la cara del cubo completa. Al final de este método se asigna la cara rotada al cubo mediante *arraycopy*.

El siguiente método se llama ***lecturaJson***y como su nombre indica recibe un archivo *json* y rellena las matrices atributo del objeto cubo con los valores que haya en el fichero json. Esto se consigue mediante los métodos paseadores diseñados por Google que nos permiten distinguir los caracteres del formato específico de los ficheros *json*. Después de detectar cada una de las matrices dentro del fichero *json* se recogen dichos valores y a través de dos bucles for anidados se recorren las matrices copiando valores de unas a otras. Al final del método se hacen llamadas a los setters de dicho objeto cubo para establecer los valores de las matrices ya una vez con los valores correctos por lo que este método devuelve *void*.

Los dos siguientes métodos van juntos ya que hacen una relación simbiótica teniendo sentido juntos, pero no separados. El primero de ellos se llama ***generateSolvedCube***que, como su nombre indica, el objetivo es generar un cubo resuelto. A través del parámetro recibido, un dato tipo int nombrado n representando el tamaño del lado, se generan matrices de dimensión nxn. En este momento entra en acción el siguiente método, ***setFace***, ya que se encarga de rellenar dichas matrices de los valores correctos. Como las caras no se representan con colores como suele ser habitual en la realidad si no que a esos colores se le asignan unos valores numéricos, una cara resuelta deberá estar representada mediante una matriz con todos los números iguales, por tanto, este método recoge por parámetro la matriz a rellenar y el número que identifica esa cara y a través de dos bucles for anidados se recorre dicha matriz asignando el valor numérico.

Al igual que ocurría en los métodos anteriores, los dos siguientes se complementan. El primer método llamado ***toString*** tiene como objetivo imprimir por pantalla el cubo desde el que se invoca este método, lo consigue concatenando cadenas de texto indicando la cara que se va a imprimir con las llamadas al siguiente método. El método que recibe dichas llamadas se llama ***faceToString***y como su nombre indica su función es imprimir por pantalla la cara que recibe por parámetro.

El método nombrado ***get\_MD5*** se encarga de hallar el código MD5 del cubo del que se invoca este método. Para ello simplemente recorre cara por cara almacenando los valores de dichas caras en una variable de tipo String para, al final del método, llamar al método ***converToMD5***que devuelve otro String a partir del que recibe por parámetro con un valor específico y representativo del mismo.

Los dos últimos métodos de esta clase, como los pares anteriores, van ligados. En este caso el método que engloba al segundo se llama ***isSolved***que, como su propio nombre indica, su cometido es comprobar si el cubo desde el que está siendo invocado está resuelto, es decir, es el cubo objetivo. Para conseguirlo primeramente creamos dos variables: la primera una variable llamada *solved* de tipo boolean, inicializada a true, y la segunda una llamada *colorseen* siendo una matriz de tipo boolean también. Seguidamente se inicializan todos los valores de la matriz *colorseen* a *false* en un for recorriendo todas sus posiciones, 6 concretamente, una posición por cada cara del cubo. En las siguientes seis líneas se hace una llamada al método ***checkSolveFace*** en el que se pasa por parámetros la cara a ser comprobada, la variable *solved* y la matriz de booleanos. En este método se recorren todos los valores de la cara en cuestión comprobando si todos los valores de la misma son iguales, lo que significaría que estaría resuelta, aunque cabe mencionar que solo comprobará si está resuelta en caso de que la cara inmediatamente anterior a ella esté resuelta. En el caso de que sí esté resuelta devolverá true y en caso contrario, false. Al retornar al método contenedor ***isSolved***almacenará en la variable *solved* el valor booleano retornado por el método ***checkSolveFace*** así, en caso de que alguna de las caras retorne false no se comprobará ninguna posterior a ella, como se ha explicado anteriormente.

* **Clase Estado**

En esta clase, como las dos anteriores, las primeras líneas de código son dos llamadas a *import* para poder utilizar los métodos de la API de Java correspondientes a la lectura de ficheros y el uso de las estructuras de datos ArrayList.

Después de definir la clase y darle su nombre, se declara su único atributo, un objeto Cubo ya que en nuestro problema no trabajaremos con cubos si no con estados que representan a los anteriores. El atributo definido por tanto será *Cubo c* y con modificador *private* para que solo pueda ser accedido desde su propia clase.

Como ya se ha mencionado en otras clases, se tiene una sobrecarga de constructores para que se pueda crear un estado a partir de un cubo dado o desde un archivo *json*.

En las siguientes 10 líneas están definidos los métodos set y get de esta clase para poder manejar el atributo a través de los mismos, respetando las convenciones de la POO.

A continuación, nos encontramos el método ***EsObjetivo***que con una llamada al método *isSolved* del *Cubo c*, devuelve un valor booleano viendo si ese estado contiene el cubo resuelto y, por tanto, es el estado objetivo. Éste método retornará el valor que retorna el método ***isSolved***.

Por último, el método probablemente con más importancia de esta clase, el método ***Sucesores*** que, a partir del cubo que contiene ese estado, genera todos los estados sucesores a él, es decir, todos los cubos resultantes de haber aplicado todos los posibles movimientos sobre él. Para conseguir esto, primeramente, inicializa una variable *n* de tipo int en la que almacena el tamaño de lado del cubo y, inicializa un ArrayList llamado ***listaSucesores***que tendrá como función almacenar todos los estados resultantes. Luego, crea dos for por cada tipo de movimiento ya que puede hacerlo en un sentido o en otro y recorre desde 0 hasta el número de lado del cubo aplicando ese movimiento a la cara concreta, creando un estado con el cubo resultante de aplicar ese movimiento y almacenando éste en la lista previamente creada. Finalmente, devuelve la lista con todos los posibles sucesores de ese estado.

* **Clase Frontera**

La clase frontera solo importa lo necesario para poder usar la Priority Queue ya que ha sido la que hemos escogido puesto que esta estructura de datos, tras comparar las distintas posibilidades, observamos que es de las más eficientes a la hora de almacenar grandes cantidades de datos y tener que ordenarlos. Estas dos características concretamente nos interesan en nuestro programa puesto que esos dos aspectos son muy importantes ya que el crecimiento de los posibles caminos para resolver el cubo que se generan es prácticamente exponencial.

Después de definir la clase se crea como atributo la propia Priority Queue *frontera* y, en su constructor, se genera un objeto *Comparator* con su método ***compare*** en el que, dado dos objetos NodoArbol (que explicaremos más adelante) se ordenan según el valor que tenga su f, de menor a mayor y, en caso de empate en los valores, se ordena según el ID que tengan, de menor a mayor.

A continuación, se han codificado los métodos ***Insertar***, ***Eliminar*** y ***estaVacia*** con sus llamadas a los métodos propios de la cola *add*, *poll* y *isEmpty*, respectivamente. Además, se ha creado un método específico llamado ***InsertarLista***para poder agregar una lista de objetos NodoArbol directamente a la frontera, recorriendo uno a uno la lista entera y añadiéndolos.

* **Clase NodoArbol**

Esta clase, al contrario de las anteriores no tiene ningún tipo de import debido a que no se utiliza ningún método de la API de Java.

Después de definir la clase, se declaran una serie de atributos: NodoArbol *nodoPadre* que tendrá la función de albergar el nodo predecesor para luego poder construir un camino solución, *static* int id que su objetivo será ir asignando un ID a cada objeto NodoArbol creado (es remarcable mencionar que es importante que la variable sea static ya que al estar dentro de la clase si no pusiésemos ese modificador no sería compartida por todos los objetos NodoArbol), Estado *estado* que será el estado que almacenará este NodoArbol que a su vez almacenará el cubo correspondiente a ese estado, int *idNodo* el ID asociado a ese NodoArbol concreto, int *costeCamino* tendrá como función ir almacenando el coste de cada acción que se aplica sobre el nodo y las anteriores a esta, String *acción* servirá para poder determinar qué acción se toma en ese nodo, int d la profundidad del mismo y por último, float *f* albergará el valor de f correspondiente a la estrategia tomada para ese problema.

Después de la declaración de los atributos se codifica el constructor asignando los valores pasados por parámetro a los distintos atributos anteriormente mencionados.

Posteriormente se generan todos los **setters** y **getters** correspondientes a esta clase para todos sus atributos y así respetar las convenciones de la programación orientada a objetos.

* **Clase Problema**

La clase problema importa el FileReader para poder abrir archivos y leerlos, en este caso de tipo json.

Después de definir la clase se crea un atributo de tipo Estado llamado *estadoInicial* que va a representar el primer estado que se nos dará como inicio y almacenará el cubo desde el que empezaremos para poder generar los sucesores y llegar a una solución.

Posteriormente se genera el constructor con sobrecarga de éste puesto que se puede recibir y crear un estado a partir de un json y a partir de un objeto Estado. Se crean los getters y setters referentes al atributo.

A continuación, nos encontramos los mismos métodos que teníamos en la clase Cubo, ***isSolved*** *y* ***chechSolveFace****,* con sus mismas funciones e implementación.

* **Clase Sucesor**

En la clase Sucesor como ocurría con la clase NodoArbol no importa nada.

Después de definir la clase se crean los atributos necesarios para representar al sucesor: int *coste* con valor 1 puesto que, por defecto, si no se nos indica ningún coste de la acción siempre va a ser 1, String *accion* para almacenar la acción que se ha provocado en ese sucesor respecto al padre y Estado *estado* para saber qué estado tiene este sucesor. Todos los atributos con modificador private para que solo puedan ser accedidos desde la propia clase y así obligar a respetar los convenios de la POO.

A continuación, nos encontramos con el constructor del objeto Sucesor que recoge por parámetro los valores de la acción y el estado del mismo. Posterior al constructor se encuentran todos los getters y setters.

* Problema a resolver:

El problema a resolver para el que se ha diseñado este programa consiste en la resolución de un cubo de Rubik de dimensión N x N x N. Esta resolución se llevará a cabo mediante las diferentes búsquedas que se han estudiado durante el curso como pueden ser A\*, profundidad, anchura, etc…

* Resolución del problema mediante los hitos planteados:
  + Hito 1

En el hito 1 nos encontramos con los siguientes objetivos a cumplir:

* Elegir un lenguaje de programación
* Construir un artefacto cubo a partir de un fichero JSON
* Crear una copia del objeto cubo en memoria

Para el primer requisito nuestro grupo eligió el lenguaje de programación Java debido que es el que más uso le hemos dado durante los cursos anteriores y por ello estamos más familiarizados con la programación en el mismo.

El segundo hito lo codificamos en la clase Cubo previamente detallada, contando con seis atributos representando cada una de las caras. En esta clase también se albergan los movimientos que se pueden aplicar al cubo. Para construirlo a partir de un fichero JSON hubo que adaptarnos a trabajar con los mismos para luego poder llegar a generar el método de lectura de dichos ficheros.

El tercer y último hito se llevó a cabo mediante el método clone de la clase anteriormente mencionada.

* + Hito 2

Los objetivos para este hito son:

* Espacio de Estados
* Problema
* Árbol de Búsqueda

Para el problema del Espacio de Estados se ha llevado a cabo la implementación de las clases Estado y Sucesor para poder cubrir todas las funcionalidades necesitadas como puede ser la generación de todos los sucesores dado un estado determinado.

La clase Problema se ha creado para poder cubrir todas las funciones a la hora de crear el problema e iniciarlo para poder resolverlo con las distintas estrategias.

El árbol de búsqueda se divide en dos clases principales, aunque está directamente relacionadas con las demás, en este caso me refiero a la clase Nodo Árbol en la que se recogen todos los valores como puede ser el ID, la f, la profundidad, nodo padre, etc… necesarios para el correcto funcionamiento del árbol siendo la clase sobre la que se va a sustentar éste. La segunda clase, pero no menos importante es la clase frontera, la cual alberga todos los nodos posibles solución ya que esta clase alberga una estructura de datos que se encarga de recoger todos los nodos con menor f, lo que los hace potenciales para formar parte de una posible solución.

* + Hito 3

El último hito recoge lo relacionado con las estrategias de búsqueda, es decir, el método para implementarlas y las clases y estructuras de datos en las que se basa este algoritmo para poder llevar a cabo su función. Básicamente, aparte de implementar en código el método de búsqueda el principal objetivo de este hito es unir todas las piezas creadas en los hitos anteriores y darle un sentido en conjunto.

* Manual de usuario:

El manual de usuario de este programa es bastante sencillo, lo primero que debemos hacer es ejecutarlo a través de la clase Búsqueda que es la que contiene el método main. A continuación, nos mostrará por pantalla un pequeño menú preguntándonos por la estrategia a seguir en ese problema concreto, previamente cargado el problema en la carpeta del proyecto. Una vez escogida la estrategia se ejecutará y después de unos pocos segundos, mostrará la solución a través de una sucesión de nodos con toda la información relativa a ellos.

La leyenda de la información, por orden de aparición, de la solución es la siguiente:

* ID del nodo.
* Movimiento que se ha aplicado respecto a su nodo padre.
* MD5 generado correspondiente a ese nodo.
* Coste acumulado para llegar a ese nodo desde el nodo inicial.
* Profundidad del nodo.
* F obtenida según la estrategia escogida.